

Environment Diagrams

Q1: Nested Calls Diagrams

Draw the environment diagram that results from executing the code below.

[See the web version of this resource for the environment diagram.](#)

HOFs

Q2: Currying

Write a function `curry` that will curry any two argument function.

```
def curry(func):
    """
    Returns a Curried version of a two-argument function FUNC.
    >>> from operator import add, mul, mod
    >>> curried_add = curry(add)
    >>> add_three = curried_add(3)
    >>> add_three(5)
    8
    >>> curried_mul = curry(mul)
    >>> mul_5 = curried_mul(5)
    >>> mul_5(42)
    210
    >>> curry(mod)(123)(10)
    3
    """
    def first(arg1):
        def second(arg2):
            return func(arg1, arg2)
        return second
    return first

# Solution with lambda
def curry_lambda(func):
    return lambda arg1: lambda arg2: func(arg1, arg2)
```

First, try implementing `curry` with `def` statements. Then attempt to implement `curry` in a single line using lambda expressions.

Recursion

Q3: Subsequences

A subsequence of a sequence S is a subset of elements from S , in the same order they appear in S . Consider the list $[1, 2, 3]$. Here are a few of its subsequences $[], [1, 3], [2]$, and $[1, 2, 3]$.

Write a function that takes in a list and returns all possible subsequences of that list. The subsequences should be returned as a list of lists, where each nested list is a subsequence of the original input.

In order to accomplish this, you might first want to write a function `insert_into_all` that takes an item and a list of lists, adds the item to the beginning of each nested list, and returns the resulting list.

```
def insert_into_all(item, nested_list):
    """Return a new list consisting of all the lists in nested_list,
    but with item added to the front of each. You can assume that
    nested_list is a list of lists.

    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    return [[item] + lst for lst in nested_list]

def subseqs(s):
    """Return a nested list (a list of lists) of all subsequences of S.
    The subsequences can appear in any order. You can assume S is a list.

    >>> seqs = subseqs([1, 2, 3])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    >>> subseqs([])
    [[]]
    """
    if not s:
        return [[]]
    else:
        subset = subseqs(s[1:])
        return insert_into_all(s[0], subset) + subset
```

OOP

Q4: Bear

Implement the `SleepyBear` and `WinkingBear` classes so that calling their `print` method matches the doctests. Use as little code as possible and try not to repeat any logic from `Eye` or `Bear`. Each blank can be filled with just two short lines.

```

class Eye:
    """An eye.

    >>> Eye().draw()
    '0'
    >>> print(Eye(False).draw(), Eye(True).draw())
    0 -
    """
    def __init__(self, closed=False):
        self.closed = closed

    def draw(self):
        if self.closed:
            return '-'
        else:
            return '0'

class Bear:
    """A bear.

    >>> Bear().print()
    ? 0o0?
    """
    def __init__(self):
        self.nose_and_mouth = 'o'

    def next_eye(self):
        return Eye()

    def print(self):
        left, right = self.next_eye(), self.next_eye()
        print('? ' + left.draw() + self.nose_and_mouth + right.draw() + '?')

```

```

class SleepyBear(Bear):
    """A bear with closed eyes.

    >>> SleepyBear().print()
    ? -o-?
    """
    def next_eye(self):
        return Eye(True)

class WinkingBear(Bear):
    """A bear whose left eye is different from its right eye.

    >>> WinkingBear().print()
    ? -o0?
    """
    def __init__(self):
        super().__init__()
        self.eye_calls = 0

    def next_eye(self):
        self.eye_calls += 1
        return Eye(self.eye_calls % 2)

```

Linked Lists

Q5: Linear Sublists

Definition: A *sublist* of linked list s is a linked list of some of the elements of s in order. For example, $\langle 3\ 6\ 2\ 5\ 1\ 7 \rangle$ has sublists $\langle 3\ 2\ 1 \rangle$ and $\langle 6\ 2\ 7 \rangle$ but not $\langle 5\ 6\ 7 \rangle$. A *linear sublist* of a linked list of numbers s is a sublist in which the difference between adjacent numbers is always the same. For example $\langle 2\ 4\ 6\ 8 \rangle$ is a linear sublist of $\langle 1\ 2\ 3\ 4\ 6\ 9\ 1\ 8\ 5 \rangle$ because the difference between each pair of adjacent elements is 2.

Implement `linear` which takes a linked list of numbers s (either a `Link` instance or `Link.empty`). It returns the longest linear sublist of s . If two linear sublists are tied for the longest, return either one.

```

def linear(s):
    """Return the longest linear sublist of a linked list s.

    >>> s = Link(9, Link(4, Link(6, Link(7, Link(8, Link(10))))))
    >>> linear(s)
    Link(4, Link(6, Link(8, Link(10))))
    >>> linear(Link(4, Link(5, s)))
    Link(4, Link(5, Link(6, Link(7, Link(8))))
    >>> linear(Link(4, Link(5, Link(4, Link(7, Link(3, Link(2, Link(8)))))))
    Link(5, Link(4, Link(3, Link(2))))
    """
    def complete(first, rest):
        "The longest linear sublist of Link(first, rest) with difference d that starts
        with first."
        if rest is Link.empty:
            return Link(first, rest)
        elif rest.first - first == d:
            return Link(first, complete(rest.first, rest.rest))
        else:
            return complete(first, rest.rest)
    if s is Link.empty:
        return s
    longest = Link(s.first) # The longest linear sublist found so far
    while s is not Link.empty:
        t = s.rest
        while t is not Link.empty:
            d = t.first - s.first
            candidate = Link(s.first, complete(t.first, t.rest))
            if length(candidate) > length(longest):
                longest = candidate
            t = t.rest
        s = s.rest
    return longest

def length(s):
    if s is Link.empty:
        return 0
    else:
        return 1 + length(s.rest)

```

Trees

Q6: Long Paths

Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a list of node labels that starts with the root and ends at a leaf. Each subsequent element must be from a label of a branch of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number

6 *Final Review*

of nodes in the path). Paths are ordered in the output list from left to right in the tree. See the doctests for some examples.

```

def long_paths(t, n):
    """Return a list of all paths in t with length at least n.

    >>> long_paths(Tree(1), 0)
    [[1]]
    >>> long_paths(Tree(1), 1)
    []
    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> print(whole)
0
  1
    2
      3
        4
        4
        5
    13
  6
    7
      8
      9
  11
    12
      13
        14
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    [0, 1, 2]
    [0, 1, 3, 4]
    [0, 1, 3, 4]
    [0, 1, 3, 5]
    [0, 6, 7, 8]
    [0, 6, 9]
    [0, 11, 12, 13, 14]
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    [0, 1, 3, 4]
    [0, 1, 3, 4]
    [0, 1, 3, 5]
    [0, 6, 7, 8]
    [0, 11, 12, 13, 14]
    >>> long_paths(whole, 4)
    [[0, 11, 12, 13, 14]]

```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

```

if n <= 0 and t.is_leaf():
    return [[t.label]]

```

Iterators & Generators

Q7: Something Different

Implement `differences`, a generator function that takes `t`, a non-empty iterator over numbers. It yields the differences between each pair of adjacent values from `t`. If `t` iterates over a positive finite number of values `n`, then `differences` should yield `n-1` times.

```
def differences(t):
    """Yield the differences between adjacent values from iterator t.

    >>> list(differences(iter([5, 2, -100, 103])))
    [-3, -102, 203]
    >>> next(differences(iter([39, 100])))
    61
    """
    last_x = next(t)
    for x in t:
        yield x - last_x
        last_x = x
```

SQL

Q8: A Secret Message

A substitution cipher replaces each word with another word in a table in order to encrypt a message. To decode an encrypted message, replace each word `x` with its corresponding `y` in a code table.

Write a select statement to decode the original message *It's The End* using the code table.

```
CREATE TABLE original AS
  SELECT 1 AS n, "It's" AS word UNION
  SELECT 2      , "The"      UNION
  SELECT 3      , "End";

CREATE TABLE code AS
  SELECT "Up" AS x, "Down" AS y UNION
  SELECT "Now"      , "Home" UNION
  SELECT "It's"    , "What" UNION
  SELECT "See"     , "Do" UNION
  SELECT "Can"     , "See" UNION
  SELECT "End"     , "Now" UNION
  SELECT "What"    , "You" UNION
  SELECT "The"     , "Happens" UNION
  SELECT "Love"    , "Scheme" UNION
  SELECT "Not"     , "Mess" UNION
  SELECT "Happens", "Go";

SELECT y FROM original, code WHERE word=x ORDER BY n;
```

What happens now? Write another select statement to decode this encrypted message using the same code table.

```
CREATE TABLE original AS
  SELECT 1 AS n, "It's" AS word UNION
  SELECT 2      , "The"      UNION
  SELECT 3      , "End";

CREATE TABLE code AS
  SELECT "Up" AS x, "Down" AS y UNION
  SELECT "Now"      , "Home" UNION
  SELECT "It's"     , "What" UNION
  SELECT "See"      , "Do" UNION
  SELECT "Can"      , "See" UNION
  SELECT "End"      , "Now" UNION
  SELECT "What"     , "You" UNION
  SELECT "The"      , "Happens" UNION
  SELECT "Love"     , "Scheme" UNION
  SELECT "Not"      , "Mess" UNION
  SELECT "Happens" , "Go";

SELECT b.y
  FROM original, code AS a, code AS b
 WHERE word=a.x AND a.y=b.x
 ORDER BY n;
```